# Design-Patterns Based Development of an Automotive Middleware

Ricardo Santos Marques, Françoise Simonot-Lion

# DESIGN-PATTERNS BASED DEVELOPMENT OF AN AUTOMOTIVE MIDDLEWARE

**Ricardo Santos Marques - Françoise Simonot-Lion**

*LORIA - INPL*
*Campus Scientifique, BP 239*
*54506 Vandoeuvre-lès-Nancy - France*
*{santos,simonot}@loria.fr*
*tel: +33 3 83 58 17 28, fax: +33 3 83 58 17 01*

Abstract: An automotive middleware layer masks the heterogeneity of platforms, and provides high level communication services to applicative tasks. In addition, this layer is a software architecture, shared between car makers and third-part suppliers, ensuring the portability and interoperability of the applicative tasks. In this study, a method aiming at developing the middleware's software architecture, and obtaining a set of tasks well characterized representing the middleware's implementation, is presented. This architecture is built with a set of design patterns, and identifies a set of middleware tasks whose characteristics allow the execution of an algorithm trying to determine a feasible priority allocation for the set of applicative and middleware tasks.

Keywords: Design Patterns, Scaling, Embedded Systems, Real-Time Middleware.

## 1. INTRODUCTION

*Context of the study.* On each node of an in-vehicle network, a set of applicative tasks execute control algorithms. Automotive functions may be performed by several distributed applicative tasks, and thus, these tasks communicate by producing and consuming signals (e.g. the number of RPM of the engine) that are sent over the network. On each node, the goal of a middleware layer is, on the one hand, to mask the heterogeneity of communication platforms. On the other hand, to offer communication services independent of applicative tasks location, and other more specialized such as diagnostic modules or I/O abstraction. In this study, the emphasis is given on the following set of communication services: sending of produced signals, and reception of signals to be consumed.

Since car makers purchase components developed by third-part suppliers, this middleware layer becomes a software architecture, shared between these actors, which ensures the portability and the interoperability of the applicative level code. Moreover, the execution of the communication services provided by the middleware interferes with the tasks running in a node, and hence, increases the probability of the timing constraint, named relative deadline, associated to the execution of applicative tasks not being met.

*Problem definition.* The problem faced by car makers and third-part suppliers is, on the one hand, the development of a middleware's software architecture that improves the maintenance and the reusability of the software components, and can be easily documented. Note that if these characteristics are achieved, the middleware's software is easily exchanged between car makers and third-

part suppliers, and can be adapted to different car makers needs. On the other hand, there is a problem of starting from this software architecture, and obtaining a middleware's implementation that allows to verify that the relative deadline imposed on the execution of tasks is respected.

*Goal of the study.* The objective of this paper is illustrated in figure 1. Precisely, it presents a method aiming at developing the middleware's software architecture, and obtaining a set of characterized tasks representing the middleware's implementation. The software architecture is composed of:

- a class diagram built from a set of design patterns, which specifies the code sequences executed to accomplish the middleware's communication services, and
- a set of tasks capable of executing on the OSEK/VDX Operating System (OSEK/VDX OS (OSEK Consortium 2005)), which is becoming the standard operating system for event-triggered automotive applications. These tasks are identified using a strategy whose criterion is adapted to the properties of OSEK/VDX OS, and implement the sequences of code identified in the class diagram.

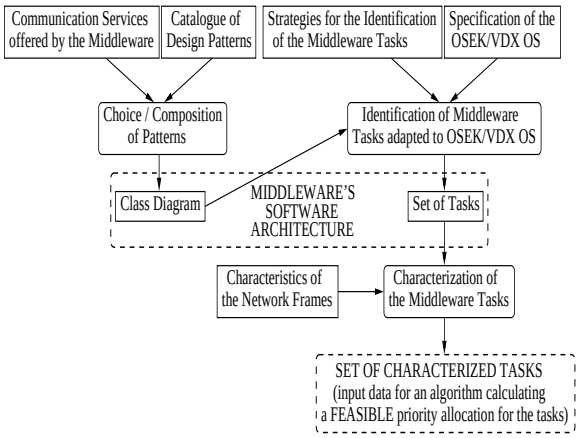


Figure 1. Objectives (represented by dashed boxes) of this study. The steps and the intermediary results are illustrated by rectangles with and without rounded corners respectively.

To obtain a set of middleware characterized tasks (characteristics like the execution time and the activation period), we use the parameters of the frames transmitted over the network. From these parameters (signals composing each frame and their emission period) we derive the work and the activation rates of the middleware tasks. From this point, one is able to quantify the interference of middleware tasks, and the entire set of tasks (applicative and middleware) form the input data for an algorithm that tries to calculate a priority allocation allowing the respect of the tasks relative deadline.

*Previous work.* To our best knowledge, design patterns (Gamma *et al.* 1995, Buschmann *et al.* 1996, Schmidt *et al.* 2000) have not been yet applied in the automotive systems development, but some work exists concerning their application to the design of a real-time middleware, TAO (**T**he **A**CE **O**RB (Schmidt and Cleeland 1999)). This middleware, specified using patterns, offers services for applications with real-time QoS requirements like video-on-demand or teleconferencing. However, it is designed to be dynamically configurable, and due to its resources consumption, is not a feasible solution in the automotive systems context, where the costs pressure is very strong. Moreover, there is no identification of the tasks that will actually implement TAO in the system.

The construction of a configuration of in-vehicle network frames has been studied in (Marques *et al.* 2003, Saket and Navet 2003). These proposed algorithms construct a set of frames, such that, the timing constraints associated to the signals are met, and the bandwidth consumption is minimized. However, these studies do not deal with the development of a middleware capable of performing the transmission and reception of these frames.

*Organization of the study.* The reminder of this paper is organized as follows: section 2 explains how the class diagram of the middleware's software architecture is obtained, and particularly, lists the set of used design patterns. Section 3 introduces the strategy allowing the identification of a set of middleware tasks able to execute on top of the OSEK/VDX OS, and whose task model permits the calculation of their interference on applicative tasks.

## 2. CLASS DIAGRAM OF THE SOFTWARE ARCHITECTURE OF THE MIDDLEWARE

A usual method for the design of software architectures is based on UML (OMG 2004). In particular, the identification of the structural components of the architecture can be achieved through the use of class diagrams (see figure 2 for an example). For this purpose, we propose a method based on design patterns, whose structural representation is done using this kind of diagram. We therefore present, on the one hand, the benefits of using design patterns for the development of the middleware's software architecture, and, on the other hand, the class diagram identifying the

components of the architecture, as well as, the design patterns used to achieve it.

### *2.1 Benefits of using design patterns*

A design pattern (Gamma *et al.* 1995, Buschmann *et al.* 1996, Schmidt *et al.* 2000) identifies the main aspects of a given object-oriented design structure: the participating classes and objects, their roles, and relations. The goal is to solve design problems arising in a certain context, to make these designs more flexible and reusable, and to improve the documentation and maintenance of existing systems by creating a pattern language. Numerous problems are addressed by design patterns: structural (architecture and organization of classes), behavioral (event-handling, synchronization, concurrency), etc.

In-vehicle embedded software, and particularly the middleware, should take advantage of the use of patterns: increased reusability and improved maintenance of software efficient solutions in order to better react to the demands of new automotive functions. Moreover, design patterns are a good solution to provide portability and interoperability between separately developed software components, which are faced with crucial issues typical of a multi-task context: concurrency and synchronization.

### *2.2 Design patterns for the software architecture*

The class diagram representing the software components of the middleware is shown in figure 2. It is composed of the set of classes that participate in the design patterns used to build the software architecture of the middleware. In order to obtain this class diagram from the set of design patterns, a "composition" activity is needed. Note that there is, for the present, no formal technique allowing to accomplish this activity. Following an intuitive rule, we selected in the structural description of each design pattern, the class that represents the core functionality of the middleware. Such a class is present in each used design pattern, and hence, we "merged" them in a unique class termed *Core* in figure 2. Obviously, the role of this class is different in each design pattern. In the following, the used patterns, as well as their application to the middleware's context, are introduced:

- *Adapter* (Gamma *et al.* 1995): this pattern allows classes to cooperate together when their interfaces are incompatible. It is composed of an abstract class defining a standard interface to be used by client classes, and of an adapter class that makes the translation between the standard interface and the incompatible one. In figure 2, this pattern is illustrated by a set of adapter classes (*AdMOST* and *AdCAN*), which adjust the interface of in-vehicle networks (MOST (MOST Cooperation 2004) and CAN (ISO 1994) in this case) to a standard set of network services defined in the abstract class *Comm*. This pattern helps the middleware to handle the heterogeneity of communication platforms, and allows the middleware's main class, named *Core*, to be developed and modified independently of the underlying communication network.
- *Observer* (Gamma *et al.* 1995): it should be used when an object must notify other objects without making assumptions about which these objects are. This pattern creates a loose dependency between objects, such that, when the state of an object changes, all its dependents (or observers) are immediately notified. It is represented in figure 2, firstly, by classes *Core* and *Comm* that must be immediately notified when a new frame arrives (class *Comm* must notify class *Core*) or is ready to be sent (class *Core* must notify class *Comm*). Secondly, by the abstract class *SubjObs* defining the interface that each observer and observed class must implement (both classes *Core* and *Comm* are "observer" and "observed"). This pattern permits classes *Core* and *Comm* to evolve independently without hindering the possibility of passing data between them.
- *Asynchronous Completion Token* (Schmidt *et al.* 2000): the purpose of this pattern is to allow an object to efficiently demultiplex the responses of asynchronous services invoked on other objects. For that, when an asynchronous service is invoked, the invoker passes a token (under the form of an object) containing information that identifies the function responsible for processing the service's response. When the service terminates, the response contains the token and thus, the invoker object can identify the function that will process the response. In the middleware's context, this pattern lets class *Core* (see figure 2) efficiently manage the frame transmission completion events dispatched by the network adapter (class *Comm* in figure 2). If the used communication platform does not provide this type of event, or the service cannot be implemented as asynchronous, the pattern can still be used with the purpose of encapsulating the information exchanged between these two actors. Hence, this pattern contributes to the creation of a loose coupling between middleware classes and still allowing an efficient exchange of data.
- *Integrated Scheduler, variant of the Active Object* (Schmidt *et al.* 2000): this pattern

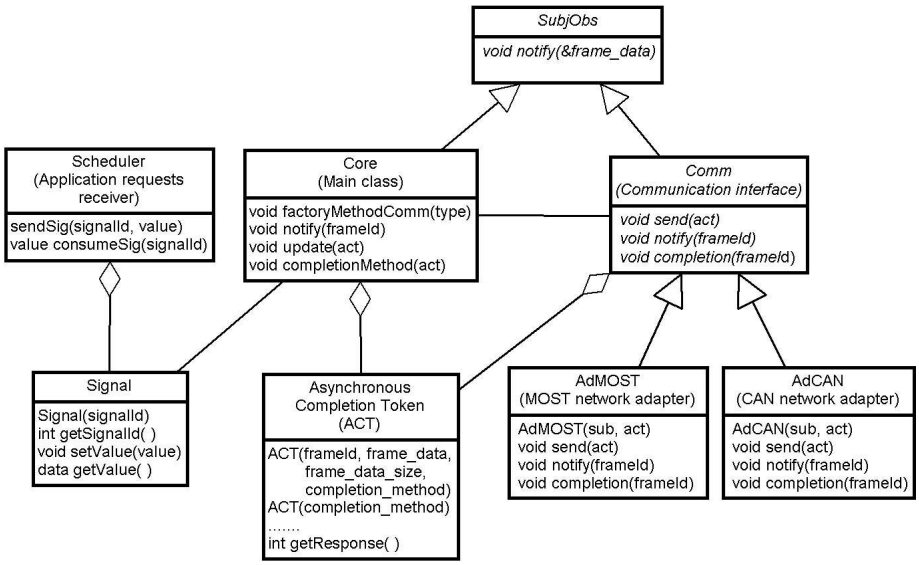


Figure 2. UML class diagram representing the software specification of the middleware. The classes are the actors of the used design patterns

addresses a concurrency aspect by decoupling the service invocation (occurring in the client's task) from the service execution (happening in a separate task). In the middleware's class diagram of figure 2 the pattern is composed of:

- a service provider represented by class *Core*,
- a service requests receiver specified by class *Scheduler* that defines the communication interface provided by the middleware, and
- a service requests repository depicted by class *Signal*, where applicative tasks store the produced signals and retrieve the signals to be consumed.

While class *Scheduler* is executed in applicative tasks, class *Core* is ran in a separate set of tasks, and class *Signal* represents a shared memory area. Therefore, the communication services provided by the middleware are executed asynchronously from applicative tasks. The main consequence is that the functionalities executed by the tasks running class *Core* simply become to, one the one hand, construct and send frames containing the produced signals, and, on the other hand, receive and handle the frames carrying the signals to be locally consumed.

Moreover, the fact that the production of signals (performed by applicative tasks) and their transmission is carried out by different tasks, has the advantage of allowing the middleware to send several signals in each frame. For this purpose, a frame packing algorithm can be used to determine a configuration containing information like the distribution of the signals among the frames, and the instants when these frames must be transmitted. Some frame packing algorithms exist applying optimization strategies aiming at, for example, minimizing the bandwidth consumption (Marques *et al.* 2003, Saket and Navet 2003).

## 3. IDENTIFICATION AND CHARACTERIZATION OF THE MIDDLEWARE TASKS

From the software architecture presented in section 2, one can determine the sequences of code that implement the services of the middleware, and can conclude that the functionalities allowing to accomplish these services are executed by a set of tasks. The next logical step is to identify this set, as well as to specify the sequence of code that will be executed by each task. Moreover, the identified tasks must be able to run on the OSEK/VDX OS, and must be characterized (activation period, execution time, ...) in order to allow the execution of an algorithm for the calculation of a feasible priority allocation for the entire set of tasks (applicative and middleware).

This section begins with the presentation of the activation mechanisms that trigger the execution of the middleware functionalities, followed by a set of strategies applicable in this context and based on the activation events (instances of the activation mechanisms) handled by the middleware. Next, the chosen strategy is introduced, the technique used to retrieve the code sequence executed by each task is given, and finally, a discussion of the chosen strategy is shown.

### *3.1 Activation mechanisms triggering the functionalities of the middleware*

The two functionalities that the set of middleware tasks must perform are the construction and sending, and the reception and handling of frames. Since these functionalities are executed asynchronously from applicative tasks (see section 2.2), the tasks supporting their execution need activation mechanisms that can be provided by the OS. OSEK/VDX OS offers different means, and among them, we keep the following: hardware interrupts, and timing alarms.

The functionality responsible for the construction and sending of frames is executed periodically according to the frame packing configuration. This functionality can be efficiently activated through a cyclic timing alarm. The execution of the functionality receiving and handling frames can be either triggered by a cyclic timing alarm (polling period), or by a network controller interrupt. The former activation mechanism degrades the middleware's performance by increasing the time delay between the arrival of the frame and its handling. Thus, in this study, we consider the following types of activation events:

- *time-triggered*: OSEK/VDX OS cyclic timing alarms for the periodic activation of the functionality responsible for the construction and transmission of frames, and
- *event-triggered*: network controller interrupts indicating the sporadic arrival of frames, and triggering the functionality in charge of receiving and handling those frames.

### *3.2 Different strategies for the identification of middleware tasks*

We have specified above the activation mechanisms for each functionality. The problem now is to determine how many tasks have to be identified according to the set of activation events. From the work of (Douglass 1999) and (Saksena *et al.* 2000), one can construct a list of strategies based on the set of activation events and applicable in the middleware's context:

(1) *one task for each event*: this strategy assigns one task for each frame that is received (if for each different frame there is a different interrupt), and for each cyclic timing alarm (assuming an alarm for each different frame emission period). The number of middleware tasks depends on the number of different frames that are received, and on the number of distinct transmission periods.

(2) *one task for each type of event*: this strategy identifies one task to handle all cyclic timing alarms, and one task to manage all network controller interrupts. The amount of middleware tasks is dependent on the different types of activation events. In this case, there are only two types and thus, two tasks.

(3) *one task for each purpose*: one example of purpose in the middleware's context is the set of signals that the nodes exchange for operation mode management (e.g. Pre-Run-Mode for node testing and network initialization, Run-Mode for full functionality of the in-vehicle system, ...). For instance, one task can be periodically activated by a cyclic timing alarm in order to send a frame containing the signal indicating the current mode, and can be activated by an interrupt caused by the arrival of the frame carrying the signal informing on the new mode. The number of middleware tasks identified by this strategy depends then on the purpose of the signals.

### *3.3 Chosen strategy*

The specification of the OSEK/VDX OS advises, according to the used conformance class, to limit to 8 or 16 the number of priorities and the number of tasks (the one executing plus those in the ready queue). Otherwise, the portability of the software components is not assured. From this limitation, one must minimize the amount of middleware tasks, allowing the execution of a maximum number of applicative tasks. One can therefore exclude the utilization of the strategies 1 and 3. The chosen strategy is the one that assigns one task to each different type of activation events. There is then one task responsible for the construction and sending of frames, activated by cyclic timing alarms, and another in charge of handling the newly arrived frames, triggered by network controller interrupts.

Note that from this point, a feasible priority allocation for the entire set of tasks (applicative and middleware) has to be determined. This can be achieved with the optimal Audsley algorithm (Audsley 1991): we recall that if a solution exists then it will necessarily be found. The feasibility test must however calculate the worst-case response time of the middleware tasks. To perform this calculation, the characteristics of the tasks are needed.

#### *3.3.1. Characteristics of the task handling frames*

In OSEK/ VDX OS, this task would be most efficiently implemented as an interrupt service routine (ISR) activated by network controller interrupts, decreasing even more the number of tasks necessary to implement the middleware. In this OS, the ISRs have a higher priority than any other regular task, hence, it is not necessary to determine its priority. To quantify its interference

on other tasks, one needs to calculate its execution time and activation period. These values depend on the type of the underlying network.

On both event-triggered and time-triggered types of networks, the time interval between the arrival of any two frames is not constant. The ISR is thus considered as sporadic (Liu and Layland 1973, Mok 1983), and its activation period is set to the smallest value possible in its context. In a event-triggered network, the activation period of the ISR is equal to the time needed to transmit the smallest frame that is received. Note that in this case, this estimation is very pessimistic. In a time-triggered context, the activation period is equal to smallest time interval between the emission of two frames received by the ISR. In both cases, the execution time is assigned to the time necessary to handle the largest frame received. The worst-case response time calculation for this type of task model is detailed in (Tindell 1992).

*3.3.2. Characteristics of the task sending frames* Being responsible for the transmission of frames, the characteristics of this task depend on the frame packing configuration. The frames to transmit can however be assigned a different emission period, and therefore, the activation rate of the task is obliged to respect all those periods. Consequently, we cannot use the usual task model where tasks have a unique activation period and execution time (Liu and Layland 1973). We have to study extended models as multiframe (Mok and Chen 1996) and generalized multiframe (GMF) (Baruah *et al.* 1999):

- if one assumes that the first emission request of all frames is issued by the first instance of the task, the multiframe task model can be used. A multiframe task $\phi_i$ is characterized by a set $C_{\phi_i}$ composed of $N$ execution times such that $C_{\phi_i} = (c_{\phi_i}^{(0)}, c_{\phi_i}^{(1)}, ..., c_{\phi_i}^{(N-1)})$, and by a unique activation period $T_{\phi_i}$ and relative deadline $\bar{D}_{\phi_i}$. The worst-case response time calculation method for this type of task model was introduced in (Takada and Sakamura 1997).

  From the frame packing configuration, one derives the characteristics of a multiframe task $\phi_i$ as follows. Let the set $Q_i = \{(Q_{f_{i,1}}, T_{f_{i,1}}), ..., (Q_{f_{i,k}}, T_{f_{i,k}})\}$ where $Q_{f_{i,k}}$ is the time needed to construct and request transmission of frame $f_{i,k}$, and $T_{f_{i,k}}$ is the transmission period of the frame. The activation period and relative deadline, $T_{\phi_i}$ and $\bar{D}_{\phi_i}$, are simply $\gcd(T_{f_{i,1}}, ..., T_{f_{i,k}})$. For each activation during the first hyperperiod, $\mathrm{lcm}(T_{f_{i,1}}, ..., T_{f_{i,k}})$, one determines the frames that are to be sent and, thus, the set of execution times for $\phi_i$:

$$\forall\, 0 \leq a \leq \mathrm{lcm}(T_{f_{i,1}}, ..., T_{f_{i,k}})/T_{\phi_i} - 1,$$

$$c_{\phi_i}^{(a)} = \sum_{\{k \,|\, a \cdot T_\phi \bmod T_{f_{i,k}} = 0\}} Q_{f_{i,k}}$$

- if one of the several execution times is greater than the relative deadline, then one can try to overcome this problem by implementing this task as generalized multiframe (GMF) (Baruah *et al.* 1999). Again, we assume that the first emission request of all frames is issued by the first instance of the task. The main difference from the multiframe task model is that the activation period and relative deadline also become a vector composed of $N$ elements. We have then $T_{\phi_i} = (t_{\phi_i}^{(0)}, t_{\phi_i}^{(1)}, ..., t_{\phi_i}^{(N-1)})$ and $\bar{D}_{\phi_i} = (\bar{d}_{\phi_i}^{(0)}, \bar{d}_{\phi_i}^{(1)}, ..., \bar{d}_{\phi_i}^{(N-1)})$. To determine the worst-case response time of GMF tasks one can use the algorithm presented in (Takada and Sakamura 1997).

  From the same set $Q_i = \{(Q_{f_{i,1}}, T_{f_{i,1}}), ..., (Q_{f_{i,k}}, T_{f_{i,k}})\}$ (see the configuration of a multiframe task), one constructs the vector of activation periods (and relative deadlines) of a GMF task $\phi_i$ as follows:

$$t_{\phi_i}^{(0)} = \min(T_{f_{i,1}}, ..., T_{f_{i,k}}),$$

$$t_{\phi_i}^{(j+1)} = \min_{\forall k}(a \cdot T_{f_{i,k}} - \sum_{l=0}^{j} t_{\phi_i}^{(l)}),$$

$$a = \min\{i \in \mathbb{N}^+ \,|\, i \cdot T_{f_{i,k}} > \sum_{l=0}^{j} t_{\phi_i}^{(l)}\}$$

  This vector is built until the following expression becomes true:

$$\sum_j t_{\phi_i}^{(j)} = \mathrm{lcm}(T_{f_{i,1}}, ..., T_{f_{i,k}})$$

  For the vector of execution times one acts in the following way:

$$c_{\phi_i}^{(0)} = \sum_k Q_{f_{i,k}},$$

$$c_{\phi_i}^{(j+1)} = \sum_{\{k \,|\, \sum_{l=0}^{j} t_{\phi_i}^{(l)} \bmod T_{f_{i,k}} = 0\}} Q_{f_{i,k}}$$

  Furthermore, since GMF tasks do not have a unique activation period, their implementation on top of OSEK/VDX OS is not trivial. Appendix A illustrates the problem and proposes a solution.

If however one of the execution times is still greater than its corresponding relative deadline, the solution is to split the work in two tasks (either multiframe or GMF). One task, having a higher priority, would be responsible for the transmission of the frames with smaller deadline,

while the other task would be in charge of sending the frames with larger deadline. This solution splits the work among two tasks, and increases the probability of respect of frames deadline, by delegating the transmission of those with a stricter timing constraint to the higher priority task.

### *3.4 Generation of the code executed by the tasks*

The sequence of code that each task must execute contributes for the task's execution time. Recall that one must well characterize each task, in order to allow the Audsley algorithm to determine the worst-case response time of each task when trying to calculate a feasible priority allocation. The code executed by each task is then retrieved by simulating the triggering of a network controller interrupt and a timing alarm, and performing a run-to-completion through the set of classes. This procedure identifies the set of objects necessary to instantiate in each task.

### *3.5 Discussion of the proposed strategy*

This strategy based on the type of the activation events, is, in our opinion, the more suited to identify a set of tasks adapted to the middleware's context. The reasons justifying this choice are:

- the minimization of the number of middleware tasks. This feature is important because OSEK/VDX OS specifies a maximum number of tasks in order to guarantee the portability of the software components;
- if new frames with different transmission periods must be sent, the characteristics of the middleware tasks change but not their task model. Nevertheless, the amount of tasks may increase if one of the execution times of the multiframe or GMF task is greater than the relative deadline (see section 3.3.2);
- if a new service is included in the middleware, these two tasks are capable of executing it if its activation events depend on interrupts or timing alarms.

Besides the maximum bound on the amount of tasks, other problems of the OSEK/VDX OS are the limit of one alarm, and the restriction of one task activated per alarm. If at least one applicative task uses the alarm, no other task, applicative or middleware, is able to employ this mechanism for its activation. An inconvenient of our strategy relies then on the fact that at least one alarm is needed (activation of the middleware task sending frames). Without any other mechanism that might be used for periodic task activation, we are obliged to take the risk of non-portability of the middleware's software.

## CONCLUSION

This study proposes a method for the development of the software architecture of an in-vehicle communication middleware. It presents a class diagram built from a set of design patterns, and introduces a strategy for the identification of a set of middleware tasks adapted to the characteristics of the OSEK/VDX Operating System.

The proposed architecture implements communication services provided to applicative tasks, masks the heterogeneity of in-vehicle networks, and specially, benefits from the advantages of using design patterns: increased reusability, improved maintenance and evolution, and easier documentation. The identified tasks accomplish the middleware's communication services, and are characterized in order to allow one to run the Audsley algorithm, aiming at determining a set of priorities that permits the respect of the relative deadline of applicative and middleware tasks.

Future work consists of the definition of an improved frame packing algorithm that tries to build the set of network frames and the priority allocation for the tasks of each node, such that, the timing constraints of applicative and middleware tasks, and signals are met. The goal is to implement the middleware's software architecture presented in this study, and to be able to generate its configuration, and the one of OSEK/VDX OS, in conformity with a given set of characterized applicative tasks and signals.

## REFERENCES


Audsley, N. (1991). Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS164. University of York.

Baruah, S., D. Chen, S. Gorinsky and A. Mok (1999). Generalized multiframe tasks. *Real-Time Systems* **17**(1), 5–22.

Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad and M. Stal (1996). *Pattern-Oriented Software Architecture - A System of Patterns.* John-Wiley and Sons.

Douglass, B. (1999). *Real-time UML Second Edition - Developing efficient objects for embedded systems.* Addison-Wesley Longman Publishing Co., Inc.

Gamma, E., R. Helm, R. Johnson and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

ISO (1994). *ISO 11898 - Road vehicles - Interchange of digital information - Controller Area Network for high-speed Communication.* International Standard Organization. ISO 11898.

Liu, C. L. and J. W. Layland (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* **20**(1), 46–61.

Marques, R. Santos, N. Navet and F. Simonot-Lion (2003). Frame packing under real-time constraints. In: *5th IFAC International Conference on Fieldbus Systems and their Applications - FeT'2003, Aveiro, Portugal.* pp. 185–192.

Mok, A. (1983). Fundamental Design Problems for the Hard Real-Time Environments. PhD thesis. Massachusetts Institute of Technology (MIT).

Mok, A. and D. Chen (1996). A multiframe model for real-time tasks. In: *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96).* IEEE Computer Society. p. 22.

MOST Cooperation (2004). MOST Media Oriented Systems Transport specification, version 2.3. Available at http://www.mostcooperation.com.

OMG (2004). *OMG Unified Modelling Language: Superstructure.* version 2.0 ed.

OSEK Consortium (2005). OSEK/VDX operating system, version 2.2.3. Available at http://www.osek-vdx.org.

Saket, R. and N. Navet (2003). Frame packing algorithms for automotive applications. Technical Report INRIA RR-4998.

Saksena, M., P. Karvelas and Y. Wang (2000). Automatic synthesis of multi-tasking implementations from real-time object-oriented models. In: *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing.* IEEE Computer Society. p. 360.

Schmidt, D. and C. Cleeland (1999). Applying patterns to develop extensible ORB middleware. *IEEE Communications Magazine.*

Schmidt, D., M. Stal, H. Rohnert and F. Buschmann (2000). *Pattern-Oriented Software Architecture.* Vol. 2: Patterns for Concurrent and Networked Objects. John-Wiley & Sons.

Takada, H. and K. Sakamura (1997). Schedulability of generalized multiframe task sets under static priority assignment. In: *RTCSA '97: Proceedings of the 4th International Workshop on Real-Time Computing Systems and Applications (RTCSA '97).* IEEE Computer Society.

Tindell, K. (1992). An extendible approach for analyzing fixed priority hard real-time tasks. Technical Report YCS189. University of York.

## Appendix A. IMPLEMENTATION OF GENERALIZED MULTIFRAME TASKS ON OSEK/VDX OPERATING SYSTEM

Since generalized multiframe (GMF) tasks do not have a unique activation period, in OSEK/VDX OS this value can only be assigned dynamically. Each instance of a GMF task, just after its beginning of execution, cancels the previous alarm, and sets a new one equivalent to the next activation period. This procedure however, does not guarantee the respect of the set of activation periods. Figure A.1 describes this problem.

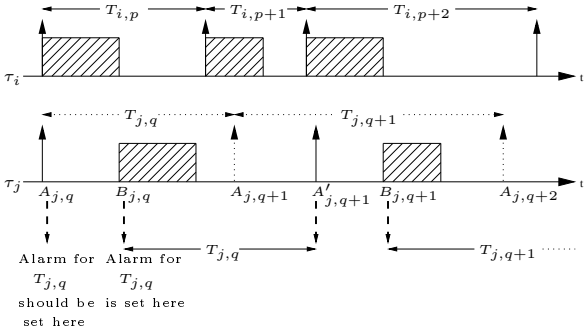


Figure A.1. This figure demonstrates the difficulty in the setting of an OSEK/VDX Operating System alarm that respects all activation periods of a GMF task. For task $\tau_j$, one could set an alarm at the activation instant of its $q$-th instance - $A_{j,q}$ - in order to activate the $(q+1)$-th instance $T_{j,q}$ units of time later. Since the $q$-th instance cannot start its execution when activated (higher priority task $\tau_i$ is running), the alarm will be set too late, at the beginning of the execution of the instance - $B_{j,q}$. The $(q+1)$-th instance is therefore activated $(B_{j,q} - A_{j,q})$ units of time too late. In the figure, instant $A_{j,q+1}$ is the activation instant that would allow the respect of $T_{j,q}$, while instant $A'_{j,q+1}$ is the activation instant that effectively occurs

Task $\tau_j$, whose $q$-th activation takes place at instant $A_{j,q}$, cannot begin its execution since an instance of an higher priority task $\tau_i$ is executing. The setting of the new alarm that should take place as sooner as possible after instant $A_{j,q}$, is effectively set at instant $B_{j,q}$ the beginning of execution of the $q$-th instance of $\tau_j$. Future activations of $\tau_j$ are now delayed of $B_{j,q} - A_{j,q}$ units of time. Note that this delay is increased each time an instance of $\tau_j$ cannot begin its execution at its activation instant. To overcome this problem, when the $q$-th instance of task $\tau_j$ begins its execution, it must calculate the value of $B_{j,q} - A_{j,q}$. Instead of setting the alarm with $T_{j,q}$, it sets with $T_{j,q} - (B_{j,q} - A_{j,q})$.